

1) Поняття паралельних (parallel), псевдопаралельних (concurrent) та розподілених (distributed) обчислень.

Паралельні обчислення передбачають одночасне виконання декількох обчислювальних завдань на різних процесорах або ядрах одного процесора. Основна мета паралельних обчислень — зменшити час виконання завдань, використовуючи кілька обчислювальних ресурсів одночасно.

Псевдопаралельні обчислення означають, що кілька завдань виглядають як такі, що виконуються одночасно, але насправді вони виконуються послідовно, але швидко переключаючись між собою. Це досягається за рахунок використання багатозадачності (multitasking) на одному процесорі.

Розподілені обчислення передбачають використання декількох комп'ютерів, об'єднаних у мережу, для виконання однієї задачі. Завдання розподіляються між комп'ютерами, кожен з яких обробляє свою частину даних і обчислень.

2) Класифікація обчислювальних систем:

S I S D

Single Instruction stream
Single Data stream

S I M D

Single Instruction stream
Multiple Data stream

M I S D

Multiple Instruction stream
Single Data stream

и

M I M D

Multiple Instruction stream
Multiple Data stream

3) Способи розпаралелювання:

- Апаратний паралелізм:
 - бітний паралелізм (bit-level parallelism): швидкість обробки збільшується за рахунок збільшення довжини машинного слова
 - на рівні інструкцій (instruction-level parallelism): швидкість обробки збільшується за рахунок конвеєрної обробки інструкцій
- Програмний паралелізм
 - паралелізм даних (data parallelism): швидкість обробки збільшується за рахунок одночасної обробки підмножин даних, що не перетинаються. MPI, OpenMP, CUDA – середовища для реалізації паралелізму даних
 - паралелізм задач обробки (task parallelism): швидкість обробки збільшується за рахунок одночасного виконання задач, що не залежать одна від одної Multithreading, Goroutines - приклади технологій, що реалізують паралелізм задач

4) Багатопоточна технологія Java

Розпиши мені багатопоточну технологію Java. Основний інструментарій, класи, перелік їхніх методів, основні високорівневі засоби тощо

Відповідь GPT4o:

Основні концепції

Багатопотоковість у Java дозволяє програмам виконувати кілька частин свого коду одночасно. Це досягається через створення та управління потоками. Ключові аспекти включають:

- **Потік (Thread):** Це незалежний шлях виконання в програмі.
- **Процес:** Запущена програма може складатися з кількох потоків.

Основні інструментарії

Клас **Thread**

Клас **Thread** представляє потік виконання в Java. Ось основні методи:

- **start():** Запускає потік.
- **run():** Метод, який визначає, що буде виконуватися в потоці.
- **sleep(long millis):** Призупиняє виконання потоку на заданий час.
- **join():** Чекає на завершення іншого потоку.
- **interrupt():** Перериває потік.
- **isAlive():** Перевіряє, чи все ще виконується потік.

Інтерфейс `Runnable`

Інтерфейс `Runnable` представляє завдання, що виконується в потоці:

- `void run()`: Метод, який визначає код завдання.

Клас `Executors`

Клас `Executors` забезпечує фабричні методи для створення пулів потоків:

- `newFixedThreadPool(int nThreads)`: Створює пул потоків з фіксованою кількістю потоків.
- `newCachedThreadPool()`: Створює пул потоків, який створює нові потоки за необхідності.
- `newSingleThreadExecutor()`: Створює пул потоків з одним потоком.

Клас `ExecutorService`

Інтерфейс `ExecutorService` забезпечує методи для керування пулом потоків:

- `submit(Runnable task)`: Виконує завдання в потоці.
- `shutdown()`: Завершує прийом нових завдань і чекає на завершення всіх завдань.
- `shutdownNow()`: Пробує зупинити всі активні завдання і повертає список незавершених завдань.

Основні високорівневі засоби

`Callable` і `Future`

- `Callable<V>`: Інтерфейс, схожий на `Runnable`, але може повертати результат і кидати винятки.
- `Future<V>`: Інтерфейс для отримання результатів асинхронних обчислень.

Методи `Future`:

- `get()`: Чекає на завершення і повертає результат.
- `cancel(boolean mayInterruptIfRunning)`: Пробує скасувати виконання.
- `isDone()`: Перевіряє, чи завершено виконання.
- `isCancelled()`: Перевіряє, чи було завдання скасовано.

Синхронізація

Ключове слово `synchronized`

Ключове слово `synchronized` забезпечує синхронізацію доступу до ресурсів між потоками.

- Синхронізація методу: `public synchronized void method() { ... }`
- Синхронізація блоку: `synchronized(this) { ... }`

Класи з пакету `java.util.concurrent`

- **ReentrantLock**: Реалізує блокування з розширеними можливостями.
- **Semaphore**: Керує доступом до ресурсу з обмеженою кількістю доступів.
- **CountDownLatch**: Чекає на зниження лічильника до нуля.
- **CyclicBarrier**: Бар'єр, який чекає, доки всі потоки не досягнуть певної точки.
- **ReadWriteLock**: Інтерфейс для реалізації блокування з роздільними замками на читання та запис.

В Java атомарність забезпечується за допомогою класів з пакету `java.util.concurrent.atomic`, які надають операції, що виконуються атомарно (тобто, безперервно і нерозривно, гарантуючи, що зміни будуть видимі іншим потокам).

Основні класи для атомарних операцій

1. `AtomicInteger`

Атомарні операції для цілих чисел.

- `get()`: Повертає поточне значення.
- `set(int newValue)`: Встановлює нове значення.
- `getAndSet(int newValue)`: Атомарно встановлює нове значення і повертає попереднє.
- `incrementAndGet()`: Атомарно збільшує значення на 1 і повертає нове значення.
- `decrementAndGet()`: Атомарно зменшує значення на 1 і повертає нове значення.
- `compareAndSet(int expect, int update)`: Якщо поточне значення дорівнює очікуваному, то встановлює нове значення.

2. `AtomicLong`

Аналогічний `AtomicInteger`, але для довгих чисел (`long`).

- `get()`: Повертає поточне значення.
- `set(long newValue)`: Встановлює нове значення.
- `getAndSet(long newValue)`: Атомарно встановлює нове значення і повертає попереднє.

- **incrementAndGet()**: Атомарно збільшує значення на 1 і повертає нове значення.
- **decrementAndGet()**: Атомарно зменшує значення на 1 і повертає нове значення.
- **compareAndSet(long expect, long update)**: Якщо поточне значення дорівнює очікуваному, то встановлює нове значення.

3. AtomicBoolean

Атомарні операції для булевих значень.

- **get()**: Повертає поточне значення.
- **set(boolean newValue)**: Встановлює нове значення.
- **getAndSet(boolean newValue)**: Атомарно встановлює нове значення і повертає попереднє.
- **compareAndSet(boolean expect, boolean update)**: Якщо поточне значення дорівнює очікуваному, то встановлює нове значення.

4. AtomicReference<V>

Атомарні операції для об'єктних посилань.

- **get()**: Повертає поточне значення.
- **set(V newValue)**: Встановлює нове значення.
- **getAndSet(V newValue)**: Атомарно встановлює нове значення і повертає попереднє.
- **compareAndSet(V expect, V update)**: Якщо поточне значення дорівнює очікуваному, то встановлює нове значення.

Додаткові атомарні класи

AtomicIntegerArray та AtomicLongArray

Атомарні операції для масивів цілих чисел та довгих чисел відповідно.

- **get(int i)**: Повертає значення елемента за індексом **i**.
- **set(int i, int newValue)**: Встановлює значення елемента за індексом **i**.
- **getAndSet(int i, int newValue)**: Атомарно встановлює значення елемента за індексом **i** і повертає попереднє.
- **compareAndSet(int i, int expect, int update)**: Якщо значення елемента за індексом **i** дорівнює очікуваному, то встановлює нове значення.

AtomicMarkableReference<V> та AtomicStampedReference<V>

Атомарні операції для об'єктних посилань з додатковими мітками та штампами.

- **AtomicMarkableReference<V>**: Зберігає посилання на об'єкт та булеву мітку.
- **AtomicStampedReference<V>**: Зберігає посилання на об'єкт та цілочисельний штамп.

5) Поняття потоку обчислень. Створення та запуск потоків в Java:

Потік – це «легкий» процес, що запускається в межах певного процесу. Потоки, що виконуються в межах одного процесу, мають спільні ресурси процесу.

Для створення потоків в Java використовують клас `java.lang.Thread`. Кожний об'єкт типу `Thread` є потоком.

Конструктор класу `Thread` приймає об'єкт, що реалізує інтерфейс `Runnable`. Клас імплементує інтерфейс `java.lang.Runnable`. Метод `run()` цього інтерфейсу призначений для опису дій, що будуть виконуватись потоком. `Runnable`-об'єкти можуть використовуватись для створення потоків, але не є потоками.

Є два типи запуску потоків у Java:

Наслідування від класу `Thread`:

```
package org.example;

public class TestClass {

    private static class MyThread extends Thread {
        private final String message;
        private final int timeOutMillis;

        MyThread(String message, int timeOutMillis) {
            this.message = message;
            this.timeOutMillis = timeOutMillis;
        }

        public void run() {
            while(true) {
                System.out.println(message);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {}
            }
        }
    }
}
```

```

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    var tOne = new MyThread("1", 600);
    var tTwo = new MyThread("2", 1300);
    tOne.start();
    tTwo.start();

    tOne.join();
    tTwo.join();
}
}

```

Реалізація кастомним класом інтерфейсу Runnable:

```

package org.example;

public class TestClassTwo {
    private static class MyRunnable implements Runnable {

        private final String message;
        private final int timeOutMillis;

        MyRunnable(String message, int timeOutMillis) {
            this.message = message;
            this.timeOutMillis = timeOutMillis;
        }

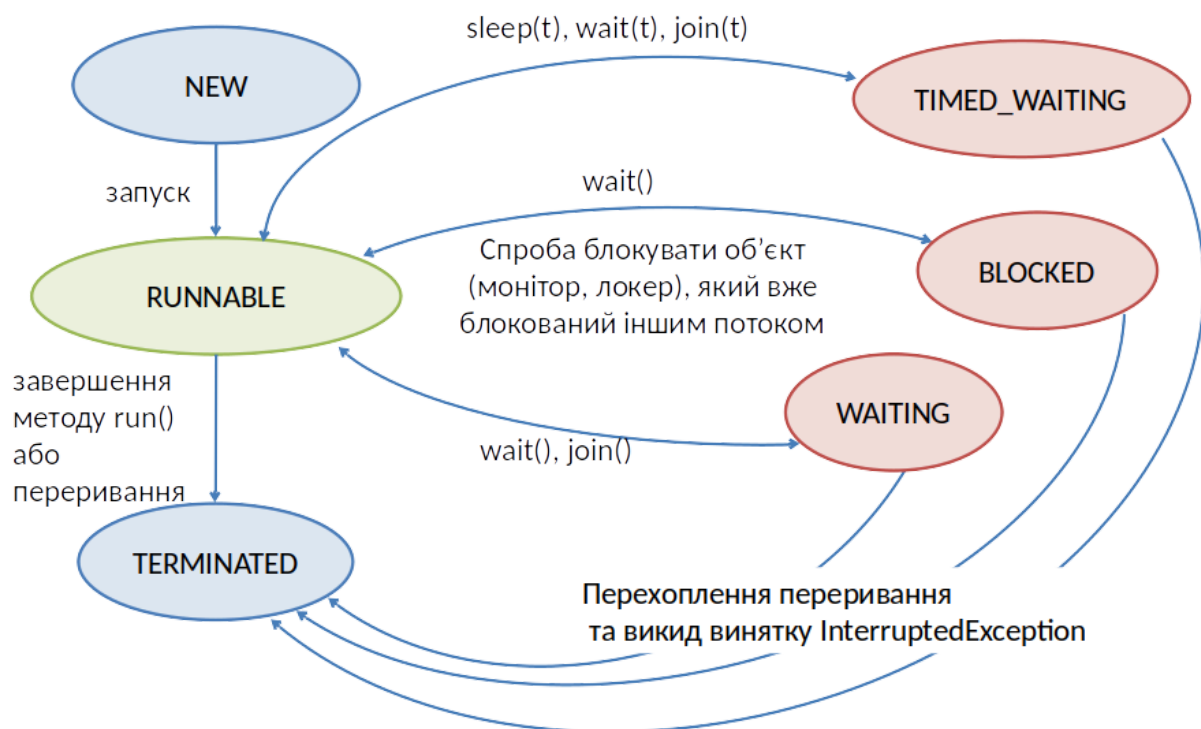
        @Override
        public void run() {
            while(true) {
                System.out.println(message);
                try {
                    Thread.sleep(timeOutMillis);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        var tOne = new Thread(new MyRunnable("1", 600));
        var tTwo = new Thread(new MyRunnable("2", 1300));
        tOne.start();
        tTwo.start();
        tOne.join();
        tTwo.join();
    }
}

```

```
}  
}
```

6) Стани потоку та переходи потоку з одного стану в інший.



Пояснення переходів

1. **NEW -> RUNNABLE:** Потік переходить з **NEW** в **RUNNABLE** після виклику методу `start()`.
2. **RUNNABLE -> BLOCKED:** Потік переходить з **RUNNABLE** в **BLOCKED**, якщо він намагається увійти в синхронізований блок або метод, але інший потік вже утримує монітор.
3. **BLOCKED -> RUNNABLE:** Потік переходить з **BLOCKED** в **RUNNABLE**, коли він отримує доступ до монітора.
4. **RUNNABLE -> WAITING:** Потік переходить з **RUNNABLE** в **WAITING** при виклику методів `Object.wait()`, `Thread.join()` або `LockSupport.park()`.
5. **WAITING -> RUNNABLE:** Потік переходить з **WAITING** в **RUNNABLE**, коли інший потік пробуджує його за допомогою методів `Object.notify()` або `Object.notifyAll()`.

6. **RUNNABLE -> TIMED_WAITING:** Потік переходить з **RUNNABLE** в **TIMED_WAITING** при виклику методів `Thread.sleep(long millis)`, `Object.wait(long timeout)`, `Thread.join(long millis)` або `LockSupport.parkNanos()`.
7. **TIMED_WAITING -> RUNNABLE:** Потік переходить з **TIMED_WAITING** в **RUNNABLE** після закінчення часу очікування або коли інший потік пробуджує його.
8. **RUNNABLE -> TERMINATED:** Потік переходить з **RUNNABLE** в **TERMINATED** після завершення виконання методу `run()`.

7) Алгоритми паралельного множення матриць.

Дивися ПИТАННЯ 8, 9

Інші паралельні алгоритми множення матриць

- Кеннона
- DNS (Dekel, Nassimi, and Sahni) алгоритм

8) Стрічковий алгоритм паралельного множення матриць.

Варіанти розпаралелення:

- 1)Одночасно розраховувати окремі групи елементів результуючої матриці
- 2)Одночасно розраховувати окремі доданки елементів результуючої матриці

Стрічкові алгоритми паралельного множення матриць:

Перший алгоритм:

кожний процес розраховує рядок результуючої матриці,
кожна ітерація процесу – один елемент рядка результуючої матриці
між ітераціями відбувається циклічна пересилка стовпчиків другої матриці
(тому і назва «стрічковий»)

У загальному випадку множення матриці А розміром $n \times s$ на матрицю В розміром $s \times m$ n процесів виконуватимуть m ітерацій обчислень за формулою:

$\forall i$ – процесу, $0 \leq i < n, j$ – ітерації, $j = i, \dots (i - 1 + m) \bmod m$

$$c_{i,j} = \sum_{k=0}^{s-1} a_{i,k} \cdot b_{k,j}$$

Другий алгоритм:

кожний процес розраховує рядок результуючої матриці,

кожна ітерація додає до кожного елемента рядка результуючої матриці доданок

У загальному випадку множення матриці А розміром $n \times s$ на матрицю В розміром $s \times m$ n процесів виконуватимуть s ітерацій обчислень за формулою:

$\forall i$ – процесу, $0 \leq i < n, k$ – ітерації, $k = i, \dots (i - 1 + s) \bmod s$

$$c_{i,j} = c_{i,j} + a_{i,k} \cdot b_{k,j}, \quad j = \overline{0, m - 1}$$

9) Алгоритм Фокса паралельного множення матриць.

1. Спочатку виконується розсилка блоків A_{ii} та B_{ij} в усі процеси P_{ij} :

$$A_{ii}, B_{ij} \rightarrow P_{ij}$$

Знаходиться їх добуток і присвоюється результуючій матриці процесу

$$C_{ij} = A_{ii} B_{ij}$$

2. На кожній ітерації:

- виконується циклічна пересилка блоків А (по другому індексу) і В (по першому індексу):

якщо $i+1 == q$, то $i+1 := 0$,

$$A_{i(i+1)}, B_{(i+1)j} \rightarrow P_{ij}.$$

- знаходиться добуток блоків, які зберігаються в процесі, і додається до результуючої матриці процесу

Таким чином,

$$C_{ij} += A_{i, (i+k) \bmod q} B_{(i+k) \bmod q, j}, \text{ де } k - \text{номер ітерації}$$

3. Результуючі матриці процесів передаються в головний процес, де з них складається результуюча матриця.

Якщо розмір матриць A і B $n \times n$, а кількість блоків по горизонталі і діагоналі q , то алгоритм реалізується на q^2 процесах, а розмір блоків матриць $m = n/q$.

10) Призупинка та відновлення роботи потоку. Переривання дії потоку.

Переривання означає термінове зупинення виконання дій потоку

Немає можливості примусово зупинити роботу потоку (метод `stop()` заборонений для використання!). Замість цього використовується метод `interrupt()`, який запитує про можливість припинення роботи даного потоку. Виклик методу `interrupt()` змінює `interrupted status` потоку на `true`. Якщо змінювання статусу переривання відбувається для методу, що викидає `InterruptedException` (наприклад, `sleep()`, `wait()`), то управління передається обробнику винятку `InterruptedException`.

Програмісту надається можливість вирішити, як потік має реагувати на зміну значення `interrupted status`. Виклик статичного методу `interrupted()` перевіряє чи змінений `interrupted status` потоку і водночас встановлює це значення в початкове, тобто `false`. Виклик методу `isInterrupted()` перевіряє, але не змінює значення `interrupted status`.

Метод `void interrupt()` надсилає запит на переривання потоку. Статус переривання змінюється при виклику цього методу на `true`. Якщо потік в момент запиту на переривання призупинений методом `wait` чи `sleep`, то генерується виняток `InterruptedException`. Виняток очищує значення статусу переривання, тобто встановлює його у значення `false`.

Метод `static boolean interrupted()` перевіряє, чи перерваний поточний потік (тобто потік, який виконує цю команду). Зверніть увагу, що виклик цього методу змінює значення статусу переривання поточного потоку на значення `false`.

Метод `boolean isInterrupted()` повертає значення статусу переривання. Цей виклик не змінює статус переривання потоку.

```
package org.example;

public class Main {

    public static class InterruptibleTask implements Runnable {

        @Override
        public void run() {

            for(var i = 0; i < 1000000; ++i) {

                System.out.printf("Iteration: %d\n", i);

                if(Thread.currentThread().isInterrupted()) {

                    System.out.println("Interrupted. Finished work.");

                    return;

                }

            }

        }

    }

    public static void main(String[] args) {

        final var t = new Thread(new InterruptibleTask());

        t.start();

        try {

            Thread.sleep(500);
```

```
    } catch (InterruptedException e) {  
  
        throw new RuntimeException(e);  
  
    }  
  
    t.interrupt();  
  
}  
}
```

11) Управління потоками в Java.

Методи `sleep()`, `wait()`, `notify()`, `notifyAll()` обробка `InterruptedException`.
Переривання потоку методом `interrupt()` Виклик статичного методу `interrupted()` перевіряє чи змінений `interrupted status` потоку і водночас встановлює це значення в початкове, тобто `false`. Виклик методу `isInterrupted()` перевіряє, але не змінює значення `interrupted status`.

Метод `static void yield()` призначений для того, щоб дати вказівку планувальнику роботи потоків про те, що в цьому місці роботи потоку виконання його дій може бути призупинено, а ресурс може бути переданий для виконання іншим потокам.

Метод `join()` примушує потік очікувати завершення роботи іншого потоку, для якого він викликаний. Потік, в методі `run()` якого був викликаний метод `join()`, зможе продовжити свою роботу тоді і тільки тоді, коли завершиться виконання потоку, що викликає метод `join()`.

Синхронізація. Якщо потрібно вказати, що виконання методу для об'єкту не може здійснюватися кількома потоками одночасно, то використовують ключове слово **`synchronized`**

12) Синхронізація в паралельних обчисленнях.

Якщо два потоки здійснюють доступ до одного і того ж об'єкта та викликають методи, що змінюють його стан, то вони «наступають один одному на п'яти» і, в залежності від порядку, в якому здійснюється доступ різних потоків до даних, інформація, яка зберігається в об'єкті, може бути неочікуваною.

1) Synchronized method:

Якщо потрібно вказати, що виконання методу для об'єкту не може здійснюватися кількома потоками одночасно, то використовують ключове слово `synchronized` в описі методу.

Наприклад,

```
public synchronized void transfer(  
    int from, int to, int amount){  
    ...  
}
```

Виклик синхронізованого методу для одного й того ж об'єкту різними потоками гарантує, що усі дії методу будуть виконуватись з об'єктом від початку до завершення тільки одним потоком.

Такий підхід запропонував Тоні Хоар (Tony Hoare).

2) Synchronized block:

```
public class Counter {  
    private Object sync = new Object();  
    private int c=0;  
    private int q=0;  
    public void incBlockSync(int k){  
        synchronized(sync){  
            c+=k;  
        }  
        q+=k;  
    }  
}
```

- 3) Об'єкт, стан якого не може бути змінений після того як він створений конструктором класу, є безпечним для спільного використання потоками.
- 4) Wait - notify
- 5) mutexes
- 6) Atomics

13) Блокування потоку

Я так розумію, що тут головне не плутати стани BLOCKED та WAITING:

BLOCKED: Потік переходить з **RUNNABLE** в **BLOCKED**, якщо він намагається увійти в синхронізований блок або метод, але інший потік вже утримує монітор.

WAITING: `Object.wait()`, `Thread.join()` або `LockSupport.park()`

TIMED WAITING: `Thread.sleep(long millis)`, `Object.wait(long timeout)`, `Thread.join(long millis)` або `LockSupport.parkNanos()`.

14) Блокування об'єкту

Кожен об'єкт має неявний локер (монітор). Кожного разу, коли потрібно забезпечити послідовний (синхронізований) доступ до полів об'єкта, потік має зробити запит на захоплення монітора об'єкта (блокування об'єкта) і отримати позитивну відповідь. Після завершення синхронізованої дії з об'єктом неявний локер розблоковується.

Якщо потік захопив монітор об'єкта, то інші потоки не можуть захопити його. При спробі захопити неявний локер об'єкта в момент, коли він заблокований іншим потоком, потік переходить у стан блокований (блокований потік) і очікує звільнення монітора

Об'єкт, для якого викликаний синхронізований метод, на весь час роботи методу блокується.

Інші потоки не мають доступу до блокованого об'єкта.

Потік, який намагається викликати синхронізований метод, перевіряє наявність блокування у об'єкта. Якщо об'єкт блокований, то потік очікує розблокування об'єкта і на весь час очікування переходить у стан Blocked.

Планувальник роботи потоків, як тільки надходить сигнал про зміну стану об'єкта, активізує потоки, що очікують його розблокування. Тільки один з потоків зможе захопити об'єкт, який звільнився від блокування.

15) Синхронізовані методи

Якщо потрібно вказати, що виконання методу для об'єкту не може здійснюватися кількома потоками одночасно, то використовують ключове слово `synchronized` в описі методу.

Наприклад,

```
public synchronized void transfer(  
    int from, int to, int amount){  
    ...  
}
```

Виклик синхронізованого методу для одного й того ж об'єкту різними потоками гарантує, що усі дії методу будуть виконуватись з об'єктом від початку до завершення тільки одним потоком.

Такий підхід запропонував Тоні Хоар (Tony Hoare).

Потік, який намагається викликати синхронізований метод, перевіряє наявність блокування у об'єкта. Якщо об'єкт блокований, то потік очікує розблокування об'єкта і на весь час очікування переходить у стан `Blocked`.

Планувальник роботи потоків, як тільки надходить сигнал про зміну стану об'єкта, активізує потоки, що очікують його розблокування. Тільки один з потоків зможе захопити об'єкт, який звільнився від блокування.

16) Проблеми управління потоками: дедлок та інші небажані стани паралельної програми

Race condition: Якщо два потоки здійснюють доступ до одного і того ж об'єкта та викликають методи, що змінюють його стан, то вони «наступають один одному на п'яти» і, в залежності від порядку, в якому здійснюється доступ різних потоків до даних, інформація, яка зберігається в об'єкті, може бути неочікуваною.

н

Взаємне блокування потоків (**Deadlock**) є наслідком застосування синхронізації виконання потоків. Потоки очікують сигналу для припинення очікування один і від одного і через це не можуть продовжити свою роботу.

Голодування потоків (**starvation**) виникає, коли через надмірну активність одних потоків інші не можуть отримати обчислювальні ресурси для виконання своїх дій.

Блокування активністю (livelock) виникає, коли потоки виконують свої дії, проте це не призводить до завершення роботи програми. Наприклад, дії одного потоку спричиняють дії іншого, які повертають перший потік на попередні дії, і так до нескінченності.

17) Високорівневі способи управління потоками з використанням бібліотеки паралельних обчислень `java.util.concurrent`.

High Level Concurrency Objects

Lock Objects

Executors

Fork/Join Framework

Concurrent Collections

Atomic Variables

Дивися питання 18, 19, 20, 21

Atomic

Операції `read/write` є атомарними для змінних посилальних типів та більшості примітивних типів (усіх окрім `long` та `double`).

Операції `read/write` є атомарними для усіх змінних, декларованих з ключовим словом `volatile` (включно з `long` та `double` змінними).

Атомарні операції не можуть перериватись іншими потоками, тому такі операції є безпечними.

`volatile` гарантує пріоритет операції `write` (по відношенню до операції `read`)

Належать класам, які забезпечують атомарні операції зі змінними

Наприклад, класи AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference<V>, AtomicIntegerArray,....

```
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
  
    public void increment() {  
        c.incrementAndGet(); // спеціальні методи класу для арифметичних  
        операцій  
    }  
  
    public void decrement() {  
        c.decrementAndGet();  
    }  
  
    public int value() {  
        return c.get();  
    }  
}
```

Concurrent Collections

!Надають можливість уникнути проблеми Memory Consistency Errors

BlockingQueue, ConcurrentMap та інші

Наприклад, використання BlockingQueue в наведеному вище прикладі дозволяє відмовитись від синхронізації методів take(), put().

18) Локери та управління потоками.

Методи інтерфейсу Lock:

lock() – блокує об'єкт, якщо той незаблокований іншим потоком, або встановлює потік у стан заблокований і примушує його очікувати звільнення локера (може повторно блокувати об'єкт, якщо локер був захоплений тим самим потоком)

tryLock() – блокує об'єкт тільки, якщо він в стані незаблокований, і надає інформацію про успішність блокування (таку інформацію можна потім обробити в програмі)

unlock() – розблоковує об'єкт, якщо він був заблокований

newCondition() – створює екземпляр класу Condition, який прикріплюється до даного Lock-екземпляру і використовується методами await(), signal() для організації очікування за умовою

Example:

```
Lock lock = ...;
```

```
lock.lock(); //блокування
```

```
try {
```

```
    // доступ до ресурсу, захищеного цим блоком
```

```
}
```

```
finally {
```

```
    lock.unlock(); // розблокування
```

```
}
```

ReentrantLock Підтримує інтерфейс Lock

Об'єкт класу ReentrantLock привласнюється потоком у разі успішного блокування і належить йому до розблокування. Блокування об'єкту відбувається успішно, якщо на момент запиту на блокування він не привласнений іншим потоком.

Потік, який є власником об'єкта, може повторно заблокувати його необмежену кількість разів. Лічильник блокувань відслідковує, щоб кількість блокувань дорівнювала кількості розблокувань.

З використанням методів `isHeldByCurrentThread()`, `getHoldCount()` можна перевірити чи є даний об'єкт привласненим потоком

Додатковий конструктор класу `ReentrantLock(boolean fair)` містить аргумент `fair`. Значення `fair`, що дорівнює `true`, встановлює, що при захопленні локера перевага надається потоку з найбільшим часом його очікування. Така «справедливість» коштує додаткових витрат часу на планування, проте запобігає «голодуванню» потоків тобто надто довгому їх очікуванню.

Клас містить набір методів, зручних для відлагодження програми: `getOwner()`, `isHeldByCurrentThread()`, `getHoldCount()`, `getQueuedThreads()`, `getWaitingThreads(Condition condition)` та інші

Example:

```
private final ReentrantLock lock = new ReentrantLock();

public void method() {

    lock.lock();

    try { // завжди використовуйте блок try/finally, щоб гарантувати виклик unlock()

        // ... method body

    }

    finally {

        lock.unlock()

    }

}
```

19) Пули потоків

Щоб зменшити накладні витрати при створенні великої кількості потоків у застосунках, використовують пули потоків

Пул потоків (Thread Pool) – це сукупність потоків, призначена для обслуговування “задач” (tasks)

Задача (Task) – об'єкт типу `Runnable`, призначений для виконання дій окремої невеличкої за обсягом підзадачі.

Основні кроки створення:

1. Створити задачі

тип `Runnable` / `Callable`

2. Створити пул потоків

тип, що підтримує інтерфейси `Executor`, `ExecutorService`: `ForkJoinPool`, `ScheduledThreadPoolExecutor`, `ThreadPoolExecutor` фабрики пулів потоків є в класі `Executors`

3. Завантажити задачі в пул: методи інтерфейсів `Executor`, `ExecutorService`: `execute()`, `submit()`, `invokeAll()`, `invokeAny()`

4. Завершити роботу пулу потоків методи інтерфейсу `ExecutorService`: `shutdown()`, `shutdownNow()`, `awaitTermination()`

5. Отримати результат виконання потоків:

як результат виконання методів `submit()`, `invokeAll()`, `invokeAny()`, що повертають результат у вигляді об'єкт `Future<T>` або список об'єктів `Future<T>`, або у вигляді звичайного об'єкту, що призначений для представлення результаті в інтерфейсі

методи інтерфейсу `Future`:

`get()`, `get(long timeout, TimeUnit unit)`, `isDone()`, `cancel()`, `isCancelled()`

20) Інтерфейс `Executor` та бібліотечні класи, які його реалізують.

Клас `Executors` містить декілька тяться також в класах `ThreadPoolExecutor` та `ScheduledThreadPoolExecutor`

Інтерфейс `Executor`

Містить єдиний метод `void execute(Runnable task)` для завантаження задач, які мають бути виконані:

```
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

Приклади реалізації методу `execute(Runnable task)`:

```
class DirectExecutor implements Executor {
    public void execute(Runnable task) {
        task.run();
    }
}
```

```

    }
}
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable task) {
        new Thread(task).start();
        // формально можливо, але не рекомендується документацією Java
        // новий потік не створюється, вирішує виконавець,
        //якому потоку доручити виконання задачі
    }
}

```

Інтерфейс `ExecutorService` extends `Executor`

Окрім методу `execute()` цей інтерфейс містить такі методи для завантаження задач :
`Future<?>`:

1. `submit(Runnable task)` – завантажує задачу в пул, метод `get` об'єкту `Future` повертає `null` у разі успішного виконання задачі, де `Future` – спеціальний тип для повернення результату виконання `run()` методу
2. `<T> Future<T> submit(Runnable task, T result)` – завантажує задачу в пул, метод `get` об'єкту `Future` повертає заданий в аргументі `result` у разі успішного виконання задачі
`<T> Future<T> submit(Callable<T> task)`
3. `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`- завантаження колекції задач, результати виконання задач повертаються, коли виконання задач завершено (успішно або через `exception`)
4. `<T> T invokeAny(Collection<? extends Callable<T>> tasks)` - завантаження колекції задач, результат повертається, коли завершено виконання однієї з задач, виконання інших при цьому відміняється.
5. `void shutdown()` – припинення завантаження нових задач: завантажені раніше задачі продовжують виконання, а при спробі завантажити нову задачу в пул буде викид `RejectedExecutionException`
6. `List<Runnable> shutdownNow()` – намагається зупинити усі задачі, які виконуються (проте без гарантії успішної зупинки), та усі, які очікують виконання, повертає список задач, які очікують виконання (при цьому завантажені раніше задачі, які не вдалось зупинити, продовжують виконання)
7. `boolean awaitTermination(long time, TimeUnit unit)` – блокування пулу доки: або завершиться виконання усіх його задач після виклику `shutdown()`, або сплине часова затримка, або дія 'current thread' перервана викликом `interrupt()`. Метод повертає значення `true` у разі успішного завершення роботи усіх потоків з пулу, `false` – у протилежному випадку, і викидає `InterruptedException` у разі отримання інформації про зміну статусу переривання.
 Документація Java рекомендує для гарантованого завершення роботи пулу потоків такий фрагмент:

```

pool.shutdown(); //заборонити надходження нових задач
try {
    // дати затримку на завершення роботи тим потокам, які раніше завантажені
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
        // припинити роботу потоків, що виконуються
        pool.shutdownNow();
        //дати затримку на очікування відповіді про успішність завершення роботи
        потоків
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Pool did not terminate");
    }
} catch (InterruptedException e) {
    // цей фрагмент на випадок якщо робота пулу потоків була перервана іншим
    pool.shutdownNow();
    Thread.currentThread().interrupt();
}

```

21) Розробка високоефективних паралельних програм з використанням ForkJoinFramework

Розміщує задачі в пул потоків ForkJoinPool extends AbstractExecutorService
 Забезпечує більшу швидкість за рахунок використання work-stealing алгоритму: потоки, які вільні, можуть здійснювати «крадіжку» роботи у потоків, які є зайнятими.
 Розроблений для алгоритмів, в яких підзадачі створюються рекурсивно

Основні класи фреймворку

```

ForkJoinPool (implements ExecutorService),
ForkJoinTask (implements Future),
RecursiveTask (implements Future),
RecursiveAction (implements Future)

```

Етапи розробки паралельного алгоритму

Задача поділяється на підзадачі, які можуть виконуватись незалежно. Підзадачі можуть розроблятися як такі, що створюють нові підзадачі.
 Для підзадач розробляється (-ються) ForkJoinTask клас (-и)
 Підзадачі (-а) додаються в пул потоків і пул запускається на виконання
 З розв'язків підзадач складається розв'язок задачі

```

static ForkJoinPool commonPool()- returns the common pool instance.
void execute(ForkJoinTask<?> task)- arranges for (asynchronous) execution of the given task.
void execute(Runnable task)- executes the given command at some time in the future.
<T> T invoke(ForkJoinTask<T> task) - performs the given task, returning its result upon completion.
void shutdown()- possibly initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

```

List<Runnable> shutdownNow() - possibly attempts to cancel and/or stop all tasks, and reject all subsequently submitted tasks.

<T> ForkJoinTask<T> submit(Callable<T> task) - submits a value-returning task for execution and returns a Future representing the pending results of the task.

<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)- submits a ForkJoinTask for execution.

<T> ForkJoinTask<T> submit(Runnable task, T result)-submits a Runnable task for execution and returns a Future representing that task.

22) Методи моделювання паралельних обчислень.

Задачі моделювання

Оцінювання ефективності паралельних обчислень застосовується у таких формах:

- Оцінювання прискорення процесу обчислень
- Оцінювання максимально можливого прискорення процесу обчислень

Перед тим, як писати код, треба взагалі подумати та прикинути, як програма буде виглядати. Найбільш ефективний спосіб це намалювати певну схему взаємодії компонентів програми. Для цього є граф “операції-операнди”, мережі Петрі.

В яких одиницях вимірює час операц

Визначення часу виконання паралельних обчислень

- Мінімально можливий час виконання паралельного алгоритму визначається довжиною максимального шляху обчислювальної схеми алгоритму:

$$T_{\infty}(G) = d(G),$$

де T_{∞} - час виконання паралельного алгоритму на **паракомп'ютері** (з нескінченно великою кількістю процесорів) $T_{\infty} = \min_p T_p$

- Мінімально можливий час виконання паралельного алгоритму, за умови, що кількість входів у кожну вершину графу не перевищує 2, обмежений знизу величиною

$$T_{\infty}(G) \geq \log_2 n,$$

де n – оцінка складності алгоритму (кількість вхідних даних, наприклад).

Визначення часу виконання паралельних обчислень

- При зменшенні кількості використовуваних процесорів час виконання алгоритму збільшується пропорційно:

$$0 < c < 1 \quad T_{cp} \leq \frac{T_p}{c}$$

Наприклад, при $p=16$, $c=0,5$: $T_8 \leq 2T_{16}$

Окремий випадок: якщо $cp = 1$, то $T_1 \leq pT_p$, або $T_p \geq \frac{T_1}{p}$.

Тобто час виконання алгоритму на p процесорах обмежений знизу величиною $\frac{T_1}{p}$.

- Часу виконання паралельного алгоритму на p процесорах обмежений зверху:

$$\forall p \quad T_p < T_{\infty} + \frac{T_1}{p},$$

де T_1 - час виконання послідовного алгоритму

Рекомендації до складання паралельних алгоритмів

- 1) При виборі обчислювальної схеми повинен використовуватись граф з мінімально можливим діаметром
- 2) Для паралельного виконання доцільно кількість процесорів визначати величиною $p \sim \frac{T_1}{T_\infty}$
- 3) Час виконання паралельного алгоритму обмежується зверху величинами:

$$\begin{aligned} \forall p \quad T_p &< T_\infty + \frac{T_1}{p} \\ \text{if } p &\geq \frac{T_1}{T_\infty} \Rightarrow T_p \leq 2T_\infty; \\ \text{if } p &< \frac{T_1}{T_\infty} \Rightarrow T_p \leq 2 \frac{T_1}{p} \end{aligned}$$

23) Модель обчислень у вигляді графа «операції—операнди»

Припущення:

- усі обчислювальні операції виконуються з однаковою з часовою затримкою рівною 1;
- передача даних між обчислювальними пристроями здійснюється миттєво (Івірно, якщо використовується спільна модель пам'яті в розподіленій системі);

Модель обчислень у вигляді графа «операції—операнди» (іноді її називають графом потоку даних) є потужним інструментом для представлення та аналізу обчислювальних процесів. У цій моделі вузли графа представляють операції, а дуги - потоки даних або операнди, які передаються між операціями.

Основні компоненти графа «операції—операнди»:

1. Вузли (Nodes):
 - Операції (Operations): Вузли, які виконують обчислення або інші операції. Наприклад, арифметичні операції (додавання, множення), логічні операції, функції тощо.

- Операнди (Operands): Вузли, які представляють значення або дані, що використовуються в операціях. Вони можуть бути змінними, константами, або результатами інших операцій.
2. Дуги (Edges):
- Дуги вказують на потік даних від одного вузла до іншого. Вони з'єднують операнди з операціями або операції з операндами, показуючи, як дані передаються та змінюються під час обчислень.

Рекомендації до складання паралельних алгоритмів

При виборі обчислювальної схеми повинен використовуватись граф з мінімально можливим діаметром

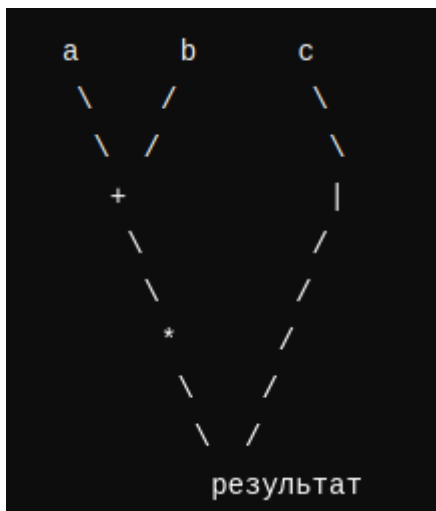
Для паралельного виконання доцільно кількість процесорів визначати величиною

Час виконання паралельного алгоритму обмежується зверху величинами:

Приклад графа «операції—операнди»:

Розглянемо простий приклад обчислення виразу: $(a+b)*c(a + b) * c(a+b)*c$.

1. Вузли:
 - Операції: додавання (+), множення (*)
 - Операнди: a, b, c
2. Граф:



24) Моделювання паралельних програм мережею Петрі.

Мережі Петрі є потужним інструментом для моделювання, аналізу та симуляції паралельних програм. Вони дозволяють чітко описувати асинхронні, розподілені та паралельні системи.

Основні компоненти мережі Петрі:

1. **Місця (Places):** Круги, які представляють умови або ресурси. Вони можуть містити маркери (tokens), що показують поточний стан системи.
2. **Переходи (Transitions):** Прямокутники або лінії, що представляють події або дії, які можуть відбутися, якщо виконані певні умови.
3. **Дуги (Arcs):** Стрілки, які з'єднують місця з переходами і переходи з місцями, показуючи потік даних або управління.

Ось чудові відео, що показують, як мережа Петрі працює на прикладі “проблеми чотирьох філософів”

▶ Dining Philosophers (2/3) - Modelled using Petri nets

▶ Dining Philosophers (3/3) - Refined using Petri nets

25) **Показники прискорення та ефективності паралельного алгоритму.

Вартість обчислень

Прискорення (Speedup)

Прискорення визначається як відношення часу виконання найкращого послідовного алгоритму до часу виконання паралельного алгоритму на p процесорах. Воно показує, у скільки разів паралельний алгоритм швидший за послідовний.

Формально, прискорення $S(p)$ визначається як:

$$S = T_{single} / T_{parallel}$$

Ефективність (Efficiency)

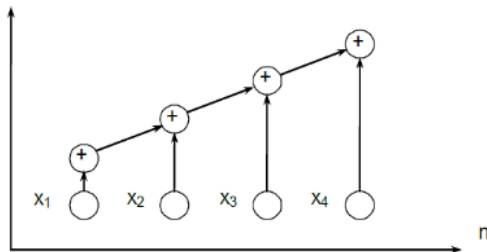
Ефективність визначається як відношення прискорення до кількості процесорів. Вона показує, наскільки добре використовуються процесори при виконанні паралельного алгоритму.

Формально, ефективність $E(p)E(p)E(p)$ визначається як:
 $E = S / n$, S - прискорення, n - кількість процесорів

26) Алгоритми паралельного сумування та оцінка їх ефективності.

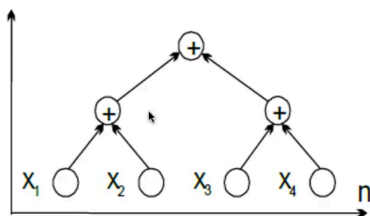
Приклад: алгоритм обчислення суми

Сума $S =$ послідовно розраховується у такі кроки: $S=0$, $S=S+x_i$ Схема цих обчислень має такий вид і не може бути розпаралелена:



Приклад: алгоритм обчислення суми

Каскадна схема паралельного алгоритму сумування:



$$T_p = \log_2 n \quad p = \frac{n}{2}$$

Ефективність алгоритму обчислення суми

Кількість операцій сумування:

$$T_1 = n - 1$$

Кількість паралельних операцій сумування:

$$T_p = \log_2 n$$

Маємо такі оцінки ефективності:

$$S_p = T_1/T_p = (n - 1)/\log_2 n,$$

$$E_p = S_p/p =$$

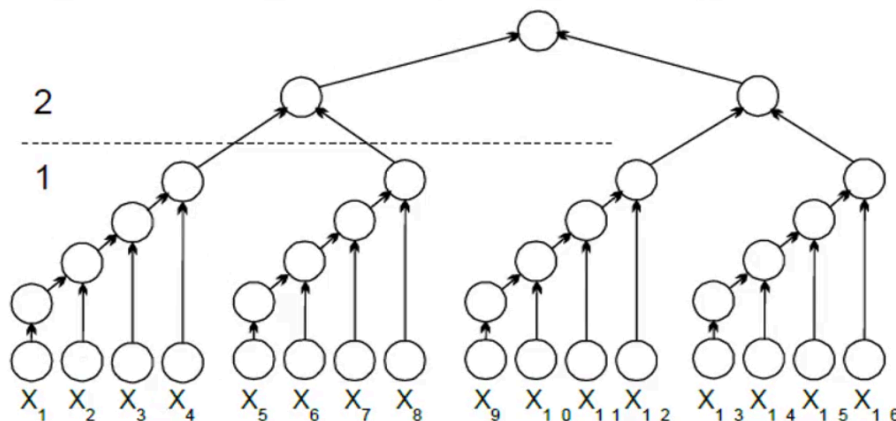
$$(n - 1)/(p \log_2 n) = (n - 1)/\left(\frac{n}{2} \log_2 n\right)$$

При зростанні кількості даних: $\lim_{n \rightarrow \infty} E_p \rightarrow 0$

Модифікована каскадна схема паралельного обчислення сум

Припустимо, що $n = 2^k$, тоді $T_\infty = \log_2 n = k$.

Кількість процесорів (кількість груп) $p = T_1/T_\infty = n/\log_2 n$



$$T_p = k + \log_2 \frac{n}{k} = \log_2 n + \log_2 \frac{n}{k} = 2 \log_2 n - \log_2 k \leq 2 \log_2 n$$

$$T_n \leq 2 \log_2 n, \quad p = \text{кількість груп} = n/k = n/\log_2 n$$

Ефективність модифікованого каскадного сумування

- $S_p = T_1/T_p = (n - 1)/(2\log_2 n)$
- $E_p = T_1/(p \cdot T_p) = (n - 1)/(n/\log_2 n \cdot 2\log_2 n) = (n - 1)/(2n)$

Маємо (!!!): $\lim_{n \rightarrow \infty} E_p \rightarrow 0,5$

27) Алгоритми паралельного сортування колекцій.

На самостійне оволодіння. Sort метод в колекціях. Показати, що знаємо кілька алгоритмів сортування та шляхів їх розпаралелювання.

GPT4o:

Arrays.parallelSort()

Метод **Arrays.parallelSort()** реалізує паралельне сортування масивів з використанням алгоритму "паралельного двійкового поділу" (Parallel Binary Merge).

Реалізація Parallel Binary Merge

Основні концепції

Алгоритм Parallel Binary Merge базується на концепції поділу і злиття, що дозволяє ефективно використовувати багатоядерні системи. Основні кроки:

1. **Розділити:** Масив ділиться на підмасиви.

2. **Сортувати паралельно:** Кожен підмасив сортується окремо в окремому потоці.
3. **Злити:** Відсортовані підмасиви зливаються в один відсортований масив.

Псевдокод алгоритму

1. Якщо масив достатньо малий, виконується послідовне сортування (наприклад, `Arrays.sort`).
2. Інакше, масив ділиться на дві частини.
3. Обидві частини сортуються паралельно (викликаючи `fork` для кожної частини).
4. Відсортовані частини зливаються за допомогою злиття двох відсортованих масивів.

Додаткові засоби для паралельної обробки

Окрім `Arrays.parallelSort()`, Java пропонує інші засоби для паралельної обробки колекцій, але вони не є методами паралельного сортування як такі. Замість цього, вони включають можливості для паралельної обробки даних, наприклад, з використанням потоків (Streams).

Паралельні потоки (Parallel Streams)

В Java 8 було додано Streams API, яке дозволяє легко обробляти дані паралельно.

Паралельні операції з потоками

- **parallelStream()**: Метод для створення паралельного потоку з колекції.
- **stream().parallel()**: Перетворення послідовного потоку в паралельний.

Приклад паралельного сортування з використанням потоків:

```
import java.util.Arrays;
```

```
import java.util.List;
```

```
import java.util.stream.Collectors;
```

```
public class ParallelStreamSortExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> list = Arrays.asList(5, 2, 9, 1, 5, 6);  
  
        List<Integer> sortedList = list.parallelStream()  
                                     .sorted()  
                                     .collect(Collectors.toList());  
    }  
}
```



```
        System.out.println(sortedList);  
    }  
}
```

28) Закон Амдала. Ефект Амдала.

Закон Амдала визначає потенційне прискорення алгоритму при збільшенні числа процесорів. Він вперше був сформульований [Джином Амдалем](#) у 1967 році^[1]. Закон стверджує, що невелика частина програми, що не піддається розпаралелюванню, обмежить загальне прискорення від розпаралелювання. Будь-яка велика математична чи інженерна задача зазвичай буде складатись з кількох частин, що можуть виконуватись паралельно, та кількох частин що виконуються тільки послідовно. Цей зв'язок задається рівнянням:

$$S = 1 / (p + (1 - p) / n)$$
, p - частка програми, що розпаралелена, n - кількість процесорів

Ефект Амдала

Ефект Амдала демонструє, що ПРИСКОРЕННЯ ЗРОСТАЄ, ОДНАК ШВИДКІСТЬ ПРИРОСТУ ПРИСКОРЕННЯ БУДЕ НАБЛИЖАТИСЯ ДО 0 ПРИ ЗБІЛЬШЕННІ КІЛЬКОСТІ ПРОЦЕСОРІВ, ТОБТО ПОХІДНА ФУНКЦІЇ У ПЕВНИЙ МОМЕНТ БУДЕ 0, А ДОТИЧНА ПАРАЛЕЛЬНА ДО ОСІ АБСЦИС.

29) Експериментальне дослідження ефективності паралельних обчислень.

кілька прогонів однакової моделі виконують одночасно для отримання більш точного результату (див. тактичне планування експериментів):

прогони виконуються в окремих потоках, основний метод очікує завершення роботи всіх потоків, потім виконує обчислення середнього значення

кілька моделей з різними параметрами виконують одночасно (див. стратегічне планування експериментів, оптимізація еволюційними методами):

основний метод здійснює обчислення параметрів, при яких виконується моделювання, та обробку результатів імітації імітація моделей для різних наборів параметрів здійснюється в окремих потоках

Графічний інтерфейс

Динамічне відтворення результатів моделювання одночасно з обчисленням моделі:

в одному потоці відбувається імітація, інший з затримкою виводить поточні значення вихідних характеристик

Анімація моделювання одночасно з обчисленням моделі:

перемальовування елементів моделі виконується з затримкою

30) Оцінка ефективності паралельних обчислень.

Експериментально та теоретично, що важливо враховувати.

Експериментально дивися ПИТАННЯ 29.

Теоретично дивися ПИТАННЯ 22, 23, 24, 25, 28

31) ***Проектування паралельних програм.

1) Чим ми керуємося, коли вибираємо той чи інший вид паралелізму? Можна на прикладі курсової розвинути тему, чим ми керувалися

2) Є ще теоретичні оцінки: закон амдала, схемами,

32) Способи підвищення продуктивності комп'ютерів. Суперкомп'ютери.

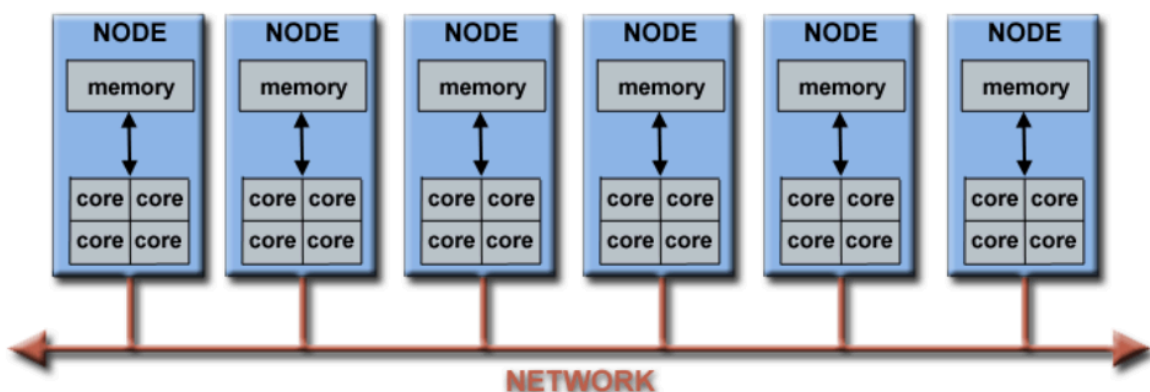
Створити розподілені обчислювальні системи, де задача одночасно використовує декілька комп'ютерів.

Розподілена обчислювальна система – набір обчислювальних вузлів (компонент), що координують свою роботу через обмін повідомленнями (синхронний чи асинхронний)

- Системи з розподіленими обчислювальними ресурсами
- Системи з розподіленою пам'яттю (розподілені бази даних)
- ❖ Хмарні обчислення
- ❖ Грід-системи

Розподілені системи. Комп'ютерний кластер

- Кожний вузол є багатопроцесорним комп'ютерним ресурсом
- Вузли з'єднані мережевим зв'язком
- Окремі вузли мають спеціальне призначення

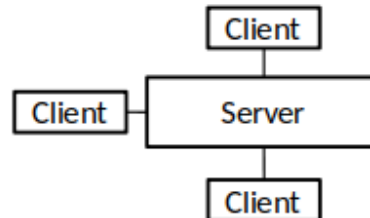


Суперкомп'ютер

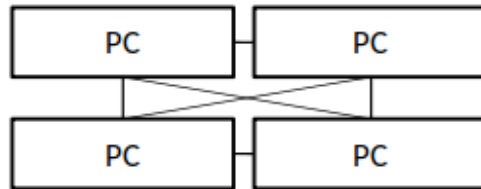
- Багатопроцесорний комп'ютер з продуктивністю набагато вищою ніж звичайний
- Найшвидші комп'ютери світу зареєстровані в TOP500 рейтингу
- Рекорд (пікової) продуктивності багатопроцесорного комп'ютерного ресурсу у 2020 році = 415,5 petaflops належить суперкомп'ютеру Fujitsu Fugaku(Japan) , другий у світовому рейтингу – суперкомп'ютер IBM Summit з продуктивністю 148,6 petaflops(US, IBM) з продуктивністю 94,6 petaflops, третій - Sierra (US, Lawrence Livermore National Laboratory) з продуктивністю 94,6 petaflops.
 - PFLOPS (peta Floating-point Operations Per Second)
- 214 суперкомп'ютерів зі списку TOP500 належать Китаю, 113 – Сполученим Штатам

Архітектура розподілених систем

- Клієнт-серверна архітектура:

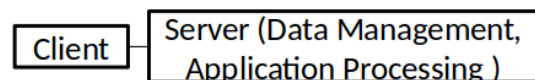


- Розподілена архітектура з рівноправними процесорами (peer to peer):

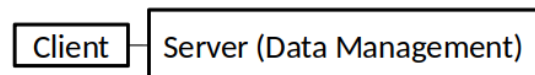


Клієнт-серверна архітектура

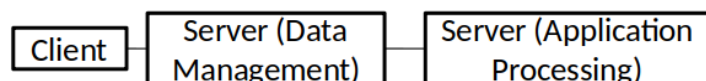
- Тонкий клієнт



- Товстий клієнт

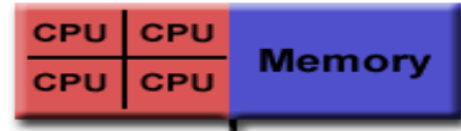


- Трирівнева архітектура

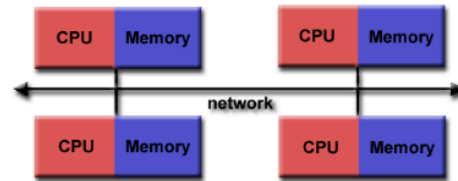


Архітектура з рівноправними процесорами

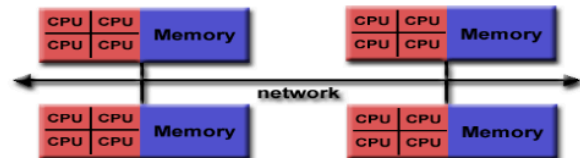
– Системи зі спільним модулем пам'яті



– Системи з розподіленими блоками пам'яті



– Гібридні системи



Таксономія Фліна: ДИВИСЯ ПИТАННЯ 2

Моделі паралельного програмування

3 точки зору взаємодії процесів:

- Спільна пам'ять
- Передача повідомлень
- Неявна взаємодія

3 точки зору розподілу на підзадачі:

- Паралелізм даних
- Паралелізм задач
- Неявний паралелізм

34) Моделі пам'яті паралельних обчислень

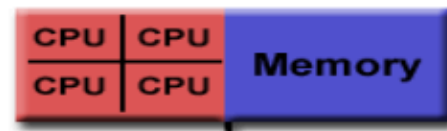
Опишу з власного досвіду:

- 1) Single memory: паралельне обчислення виконується в одному процесі, але в декількох його потоках; усі потоки знаходяться у спільному адресному просторі, тому можуть взаємодіяти з ресурсами один одного;
- 2) Shared memory: паралельне обчислення виконується в декількох процесах, ці процеси мають якимось розділяти спільний ресурс, тоді вони мають у свій адресний простір спільну фізичну пам'ять. Далі вони вже редагують цю пам'ять, і це можуть бачити інші процеси
- 3) Гібридна: паралельне обчислення використовує обидва варіанти

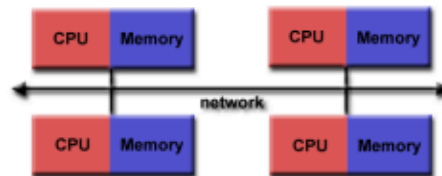
Ось навіть слайд у І.В. Стеценко є з чимось схожим:

Архітектура з рівноправними процесорами

– Системи зі спільним модулем пам'яті



– Системи з розподіленими блоками пам'яті



– Гібридні системи



35) Стандарти та технології обміну повідомленням в розподілених системах.

Існують різні реалізації стандарту MPI (ПИТАННЯ 36) та CORBA. Далі дописати це питання

CORBA (Common Object Request Broker Architecture)

Огляд

CORBA — це стандарт, розроблений Object Management Group (OMG), який дозволяє об'єктам, розташованим в різних середовищах, взаємодіяти один з одним. CORBA забезпечує платформно-незалежний механізм для віддаленого виклику методів.

Основні характеристики

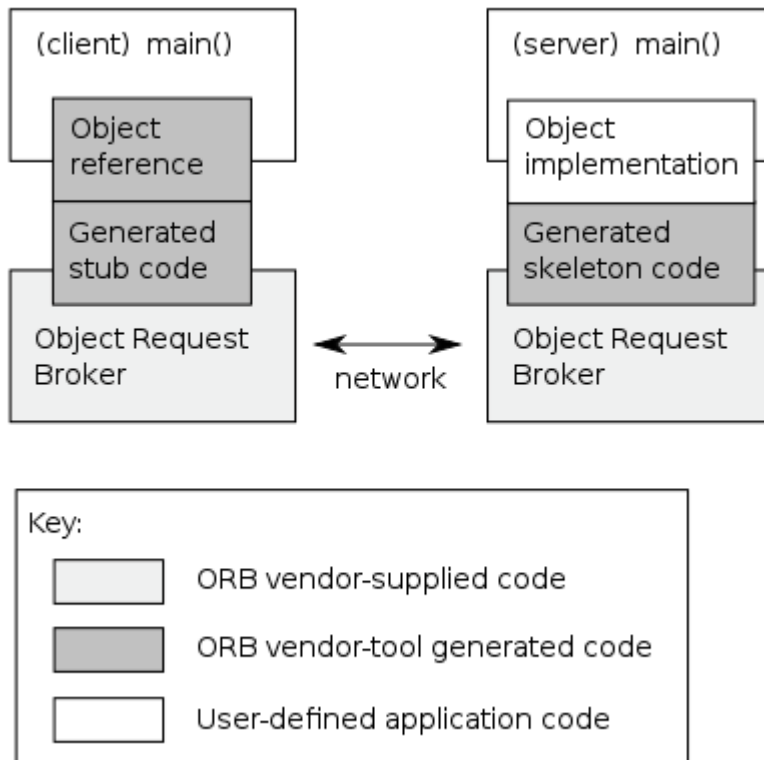
1. **ORB (Object Request Broker)**: Центральний компонент, який обробляє всі запити на виклик методів від одного об'єкта до іншого.
2. **IDL (Interface Definition Language)**: Мова для визначення інтерфейсів об'єктів, які можуть взаємодіяти через CORBA.
3. **Незалежність від мови програмування**: CORBA підтримує багато мов програмування, включаючи C++, Java, Python тощо.

Основні компоненти

- **ORB (Object Request Broker)**: Посередник для обміну запитами між об'єктами.
- **IDL (Interface Definition Language)**: Використовується для визначення інтерфейсів об'єктів.
- **POA (Portable Object Adapter)**: Управляє об'єктами-сервантами і забезпечує їхню життєдіяльність.

https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture

In order to build a system that uses or implements a CORBA-based distributed object interface, a developer must either obtain or write the IDL code that defines the object-oriented interface to the logic the system will use or implement. Typically, an ORB implementation includes a tool called an IDL compiler that translates the IDL interface into the target language for use in that part of the system. A traditional compiler then compiles the generated code to create the linkable-object files for use in the application. This diagram illustrates how the generated code is used within the CORBA infrastructure:



36) Стандарт Message Passing Interface (MPI)

<https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>

Стандарт Message Passing Interface (MPI) - це специфікація для розробників та користувачів бібліотек передачі повідомлень. Стандарт MPI не є бібліотекою, це тільки опис того, що має бути в такій бібліотеці.

MPI відтворює модель паралельного програмування “передачі повідомлень”: паралельні процеси здійснюють обмін повідомленнями (даними) за допомогою комунікаційних операцій.

Метою стандарту Message Passing Interface є забезпечення достатньо широкого стандарту для написання програм передачі повідомлень. Інтерфейс має бути: практичним, портативним, ефективним, гнучким

Остання версія MPI: MPI-4.

Специфікації інтерфейсу визначені у прив’язці до мов програмування C та Fortran90. MPI-бібліотеки мов програмування можуть відрізнятися в залежності від версії стандарту MPI, що підтримується. Розробникам/користувачам потрібно це враховувати.

Open MPI – потокобезпечна, відкрита реалізація стандарту MPI мовою C. Open MPI доступний для використання на більшості Linux кластерів (LC). Вам потрібно завантажити бажаний пакет інструментів з використанням use команди. Наприклад, use -l (список доступних пакетів) use openmpi-gnu-1.4.3 (використовувати обраний пакет)
Це гарантує, що сценарії обгортки MPI LC вказують на потрібну версію Open MPI. Більше інформації про Open MPI доступно за посиланням www.open-mpi.org

37) Методи обміну повідомленнями стандарту MPI один до одного та їх застосування для розробки паралельних програм.

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are send and receive.

Chapter 3

Point-to-Point Communication

3.1 Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism.

The basic point-to-point communication operations are send and receive. Their use is illustrated in the example below.

```
#include "mpi.h"
int main( int argc, char *argv[])
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0) /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99,
        MPI_COMM_WORLD);
    }
    else if (myrank == 1) /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

```
    return 0;  
}
```

In this example, process zero (myrank = 0) sends a message to process one using the send operation **MPI_SEND**. The operation specifies a send buffer in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable message in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send Operation.

The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (myrank = 1) receives this message with the receive operation **MPI_RECV**. The message to be received is selected according to the value of its envelope, and the message data is stored into the receive buffer.

Методи передачі повідомлень один-до-одного

MPI методи передачі повідомлень point-to-point, як правило, мають такий список аргументів:

Blocking sends	MPI_Send(buffer,count,type,dest,tag,comm)
Non-blocking sends	MPI_Isend(buffer,count,type,dest,tag,comm,request)
Blocking receive	MPI_Recv(buffer,count,type,source,tag,comm,status)
Non-blocking receive	MPI_Irecv(buffer,count,type,source,tag,comm,request)

38) Колективні методи обміну повідомленнями стандарту MPI та їх застосування для розробки паралельних програм

MPI_Barrier

Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.

MPI_Barrier (comm)

MPI_BARRIER (comm,ierr)

MPI_Bcast

Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

MPI_Bcast (&buffer,count,datatype,root,comm)

MPI_BCAST (buffer,count,datatype,root,comm,ierr)

MPI_Scatter

Data movement operation. Distributes distinct messages from a single source task to each task in the group.

MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,root,comm)

MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf, recvcnt,recvtype,root,comm,ierr)

MPI_Gather

Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf, recvcount,recvtype,root,comm)

MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf, recvcount,recvtype,root,comm,ierr)

MPI_Allgather

Data movement operation. Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.

MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf, recvcount,recvtype,comm)

MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf, recvcount,recvtype,comm,info)

MPI_Reduce

Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.

MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)

MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)

39) OpenMPI: особливості побудови програми та основні методи.

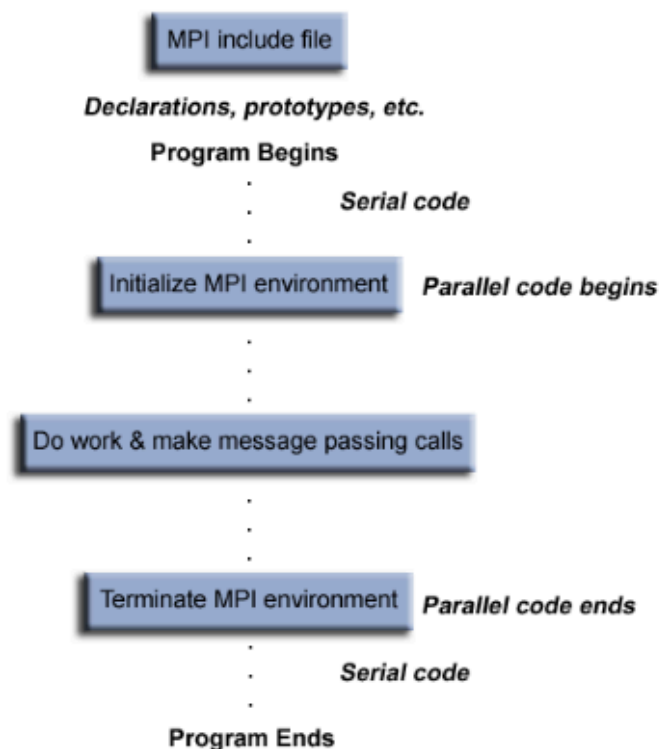
OpenMPI – це бібліотека реалізації стандарту MPI (Message Passing Interface), яка використовується для паралельних обчислень в розподілених системах. OpenMPI забезпечує засоби для обміну повідомленнями між процесами, синхронізації та управління процесами.

Особливості побудови програми з використанням OpenMPI

1. Ініціалізація та завершення програми:

- Кожна MPI-програма починається з ініціалізації MPI-середовища і завершується його завершенням.
 - Використовуються функції `MPI_Init` та `MPI_Finalize`.
2. **Комунікатор:**
- Комунікатори є ключовою концепцією MPI, які визначають групу процесів, що можуть обмінюватися повідомленнями.
 - Основний комунікатор `MPI_COMM_WORLD` включає всі процеси.
3. **Ранги процесів:**
- Кожен процес у комунікаторі має свій унікальний ідентифікатор, званий рангом.
 - Ранг використовується для адресації процесів під час обміну повідомленнями.
4. **Обмін повідомленнями:**
- Використовуються різні методи для відправлення та приймання повідомлень, як блокуючі, так і неблокуючі.
 - Методи включають точковий обмін повідомленнями (`MPI_Send`, `MPI_Recv`), колективні операції (`MPI_Bcast`, `MPI_Reduce`), синхронізацію (`MPI_Barrier`) тощо.

Загальна структура MPI програми



Основні методи OpenMPI

MPI_Init

Ініціалізує середовище виконання MPI. Ця функція повинна бути викликана у кожній програмі MPI перед викликом будь-яких інших функцій MPI і повинна бути викликана тільки один раз в програмі MPI.

MPI_Init (&argc,&argv) // Open MPI

MPI_INIT (ierr) // FORTRAN

MPI_Initialized

Указує, чи був викликаний MPI_Init - повертає логічне значення true (1) або false(0). За стандартом MPI MPI_Init може бути викликаний один і тільки один раз кожним процесом.

MPI_Initialized (&flag)

MPI_INITIALIZED (flag,ierr)

MPI_Finalize

Завершує середовище виконання MPI. Ця функція повинна бути останньою процедурою MPI, яка викликається в кожній програмі MPI - ніякі інші процедури MPI не можуть бути викликані після неї.

MPI_Finalize ()

MPI_FINALIZE (ierr)

Інші методи можна переглянути у питаннях 37, 38

40) ****Розробка ефективних паралельних алгоритмів в OpenMPI.**

Цитата І.В.Стеценко:

“Почати з оцінювання ефективності паралельної програми, продовжити описом що впливає на ефективність, далі в кожній з вивчених технологій вказати, які механізми дозволяють найбільш повновикористати обчислювальний ресурс. Тому це питання більш загальне, ніж про OpenMPI.”

Акцентувати увагу на методах, з точки зору ефективного використання:

Тобто чому саме доречно використати MPI_Gather, MPI_Reduce, MPI_Send, MPI_Recv. Можна навести приклад з курсової чи практикумів.

41) **Оцінювання продуктивності паралельних алгоритмів в розподілених системах

Написати про флопси продуктивності, далі ще є затримки на передачу повідомлень в MPI Express. ТУТ НЕ ЗВУЧАТЬ ПОКАЗНИКИ ПРИСКОРЕННЯ ТА ЕФЕКТИВНОСТІ.

Потужність розподілених систем (суперкомп'ютерів)

Потужність розподілених систем зазвичай вимірюється в таких одиницях: **FLOPS (Floating Point Operations Per Second)**: Кількість операцій з рухомою комою, які система може виконати за секунду.

42) Технологія Remote Method Invocation (RMI).

RMI є розподілена об'єктна система, що надає можливість легко розробляти розподілені Java-додатки.

Розробка розподілених додатків в RMI простіша, ніж розробка з сокетами, оскільки немає необхідності в розробці протоколу.

В RMI, розробник має ілюзію виклику локального методу з локального файлу класу, в той момент як насправді аргументи відправляються до віддаленого ресурсу та інтерпретуються, а результати відправляються назад замовнику.

Розробка розподілених додатків з використанням RMI складається з таких кроків:

- визначити віддалений інтерфейс
- реалізувати віддалений інтерфейс
- розробити сервер
- розробити клієнт
- створити client stub і server skeleton, запустити RMI реєстрацію серверу і клієнта

Remote Method Invocation (RMI)

RMI є розподілена об'єктна система, що надає можливість легко розробляти розподілені Java-додатки. Розробка розподілених додатків в RMI простіша, ніж розробка з сокетами, оскільки немає необхідності в розробці протоколу.

В RMI, розробник має ілюзію виклику локального методу з локального файлу класу, в той момент як насправді аргументи відправляються до віддаленого ресурсу та інтерпретуються, а результати відправляються назад замовнику.

43) Грід-технології. Поняття спільного віртуального ресурсу. Програмне забезпечення грід-технологій.

Грід-системою є вільне об'єднання обчислювальних та, можливо, інформаційних ресурсів користувачів.

Grid є технологією забезпечення гнучкого, безпечного і скоординованого загального доступу до ресурсів

Технологія Grid не є технологією паралельних обчислень, в її завдання входить лише координація використання ресурсів

Типи грід-систем

- обчислювальні грід-системи (Computing Grid),
- інформаційні грід-системи (Data Grid)
- мішані грід-системи

Поняття спільного віртуального ресурсу

Спільний віртуальний ресурс (віртуальна організація) – це сукупність обчислювальних, мережових і інформаційних ресурсів, які надаються користувачам через грід-інфраструктуру. Користувачі можуть прозоро використовувати ці ресурси для виконання своїх обчислювальних задач, не турбуючись про фізичне розташування або характеристики окремих вузлів.

Основні компоненти спільного віртуального ресурсу:

1. **Обчислювальні ресурси:**
 - Сервери, кластери, суперкомп'ютери, що надають потужності для виконання обчислень.
2. **Зберігання даних:**
 - Дискові масиви, файлові системи, бази даних, що забезпечують зберігання та доступ до великих обсягів даних.
3. **Мережеві ресурси:**
 - Високошвидкісні мережі, що забезпечують комунікацію між різними вузлами ґрід-у.
4. **Програмне забезпечення:**
 - Інструменти та середовища для управління ресурсами, планування задач, моніторингу та забезпечення безпеки.

Програмне забезпечення для ґрід-систем

- Globus Toolkit
- Sun Grid Engine
- Sun Grid Engine, Enterprise Edition (комерційне)

Платформа – взаємоузгоджений набір засобів, здатних дати комплексне рішення задачі обслуговування Grid - інфраструктури виробничого призначення

- ARC(NorduGrid),
- gLite (www.glite.org),
- gUSE (grid User Support Environment), gUSE / WSPGRADE
- Alien,
- LCG,
- DataGrid (www.datagrid.org),
- Unicore (www.unicore.org),
-
- В основі всіх - Globus Toolkit (www.globus.org)

44) Організація і управління розподіленими ґрід-ресурсами

Архітектура ґрід-системи

Складові архітектури:

- Метапланувальник
- розподілені обчислювальні ресурси (спільний віртуальний ресурс користувача)

Види архітектур:

- Однорівнева: планувальник – комп'ютери користувачів

- Дворівнева: метапланувальник – локальні планувальники - комп'ютери користувачів

Архітектура дворівневої грід-системи



Метапланувальник

Метапланувальник здійснює розподіл завдань між обчислювальними кластерами

У розподілених обчислювальних вузлах та сховищах даних встановлені грід-сервіси програмної інфраструктури, що надають інформацію про поточний стан ресурсу планувальнику виконання завдань

Метапланувальник взаємодіє не з апаратними ресурсами, а з грід-сервісами, що представляють ці ресурси

Локальні планувальники забезпечують управління виконанням завдань локальним обчислювальним ресурсом і взаємодіють з метапланувальником

Управління грід-ресурсами

Задача управління грід-ресурсами полягає в тому, щоб на основі інформації про поточний стан розподілених обчислювальних ресурсів та сховищ даних надавати дозвіл на використання віртуального обчислювального ресурсу користувачам грід-системи в залежності від їх потреби та у відповідності до їх вкладу у віртуальний обчислювальний ресурс (або пріоритету використання ресурсу), а також надавати несуперечливий доступ до інформаційного ресурсу.

Політики розподілу ресурсів

Sun Grid Engine, Enterprise Edition (SGEEE) надає можливість вибору однієї з чотирьох політик розподілу ресурсів між незалежно працюючими користувачами:

- політика розділених ресурсів,
- функціональна політика,
- політика „роботи до строку”,
- політика явного виділення ресурсів

В усіх запропонованих політиках користувач отримує дозвіл на використання ресурсу в залежності від наданої йому за домовленістю квоти на використання ресурсу, що визначається часткою віртуального ресурсу, на яку претендує користувач, або пріоритетом користувача на використання ресурсу

Розподіл ресурсів планувальником SGEEE

Розподіл ресурсів здійснюється планувальником на визначений інтервал часу.

Планувальник розподіляє ресурси, виходячи з співвідношення білетів, „закуплених” на даний інтервал часу.

Користувачі „купують квитки”, тобто посилають планувальнику замовлення ресурсу, для того, щоб отримати дозвіл на використання ресурсу у наступний інтервал часу.

Планувальник, виходячи з одержаних замовлень на використання ресурсів, здійснює розподіл ресурсів.

Для кожного користувача підраховується усереднене фактичне використання обчислювального ресурсу і якщо воно перебільшує (або зменшує) надану квоту, то спрацьовує механізм компенсації пере- (недо-) споживання обчислювального ресурсу. Тобто, якщо користувач певний час не використовував ресурс, то його компенсаційний коефіцієнт збільшився і він має право отримати більшу частину ресурсу.

Якщо завдання, які захопили ресурс (раніше інших завдань), використовують ресурс більше наданої квоти (більше, ніж надає квиток), то користувач лишається права отримати певну кількість квитків.

